AN4678
Application note

Full duplex SPI emulation for STM32F4 microcontrollers

Introduction

The STMCube™ initiative was originated by STMicroelectronics to ease developers life by reducing development efforts, time and cost. STM32Cube covers the STM32 portfolio.

STM32Cube Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows to generate C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (namely, STM32CubeF4 for STM32F4 series)
  - The STM32Cube HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio
  - A consistent set of middleware components such as RTOS, USB, TCP/IP and graphics
  - All embedded software utilities coming with a full set of examples.

This Application Note describes how to implement a Serial Peripheral Interface (SPI) emulator for the microcontrollers of the STM32F4 series.

An SPI interface is commonly emulated in software where a dedicated hardware peripheral is not available. It is also needed in applications that require more SPIs than those offered by STM32F4 microcontrollers. Using this software the user can compensate the limited number of SPI peripherals, without the need to switch to higher level MCUs with sufficient number of SPIs when the application doesn’t require additional performance and functionality.

This SPI is full-duplex, supports 8, 16 data length bits and clock speed up to 6 MHz with CPU operating at 168 MHz. It also offers a high flexibility since any I/O pin can be configured as Master-Out/Slave-In (MOSI) and Master-In/Slave-Out (MISO). In addition, this SPI emulation uses the DMA to minimize the software overhead and CPU load, which may significantly impact the system ability to execute other tasks and to meet real-time schedules.

This application note provides a basic example of communication between a hardware and a software SPI as well as a summary of CPU load and firmware footprint.

A firmware package (X-CUBE-SPI-EMUL) is delivered with this document and contains the source code of the SPI emulator with all the drivers needed to run the example.

Contents

List of tables

List of figures

1 SPI emulator description

This section describes the implementation of an SPI emulation by defining the system level requirements.

1.1 Main features

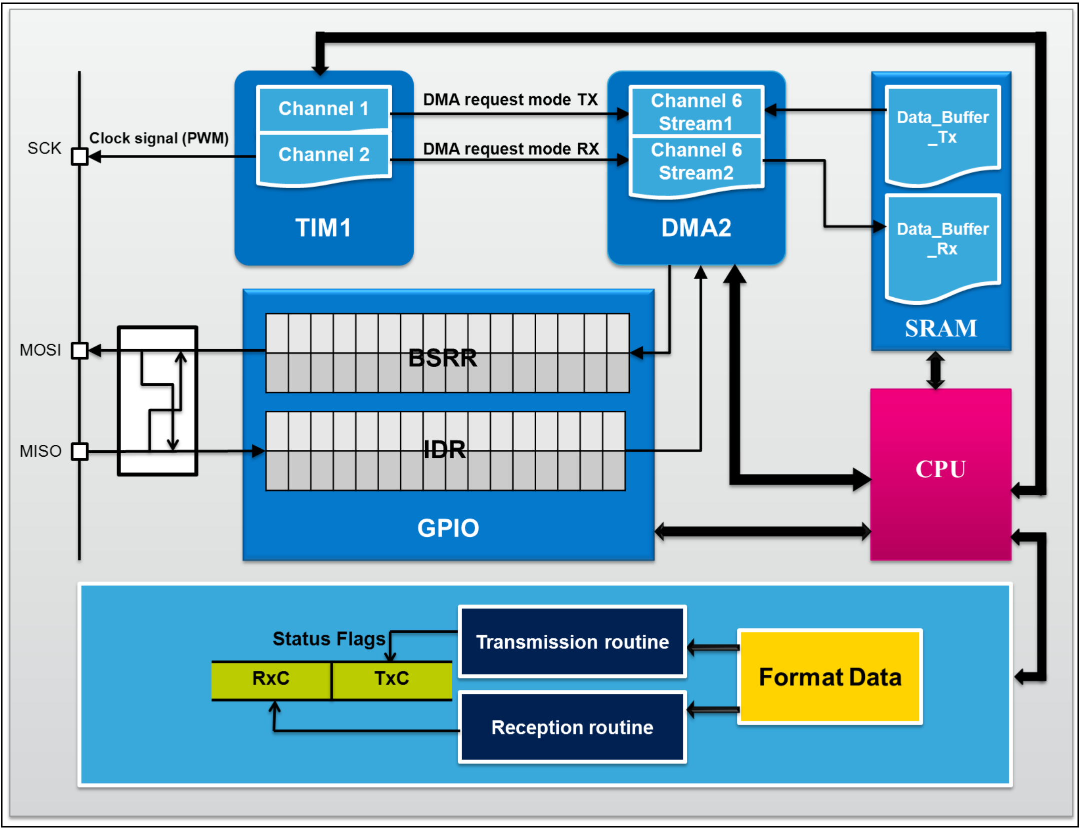
The main features of the SPI emulator are:

- Simplex/ full-duplex, synchronous, serial communication
- Master and slave operations
- SPI clock up to 12 MHz in simplex mode with CPU operating at 168 MHz
- SPI clock up to 6 MHz in full-duplex mode with CPU operating at 168 MHz
- Programmable data word length: 8 and 16 bits
- Programmable clock polarity and phase
- Programmable data order with MSB-first or LSB-first shifting
- Flexible GPIO usage: all GPIOs can be configured as SPI MOSI/MISO
- Status flags/interrupt
  - Transmit Complete (TxC)
  - Receive Complete (RxC)

1.2 SPI emulator block diagram

Figure 1 gives an overview of the interaction between the hardware peripherals and the software modules that make up the SPI emulator.

Figure 1. SPI emulator block diagram



The SPI emulator implementation is based on GPIO, timer and DMA peripherals.

- Three lines are used to connect the SPI emulator to external devices. Data are transmitted to the BSRR and IDR registers in Tx and Rx mode respectively.
- Data transfers are performed by DMA2 with two dedicate channels:
  - Channel6 Stream1 for data transfers in Tx mode
  - Channel6 Stream2 for data transfers in Rx mode
- Timer overflow and IO capture compare events are used to control timing of SPI emulator input sampling and output handling of the Rx and Tx signals. TIM1 generates the clock signal in master mode, and sends requests to DMA to transfer data at the required speed for both master and slave mode.

SPI emulator peripherals requirements and configurations are described in *Table 1*.

Table 1. SPI emulator peripherals requirements and configurations

Mode			Master		Slave	
			Tx	**Rx**	**Tx**	**Rx**
Peripheral	**TIM1**	Channel	Channel1	Channel2	Channel1	Channel2
		Configuration	PWM Mode (Capture Compare)			
	DMA2	Channel	Channel6 Stream1	Channel6 Stream2	Channel6 Stream1	Channel6 Stream2
		Configuration	Memory to peripheral	Peripheral to memory	Memory to peripheral	Peripheral to memory
	GPIO	MOSI	Any I/O configured as output		Any I/O configured as input	
		MISO	Any I/O configured as input		Any I/O configured as output	
		SCK	PA8 or PA9 configured in AF			

Note: *The peripherals required for full duplex communication are the combination of peripherals used in Tx mode and Rx mode.*

Note: *Other Timers with the same features as TIM1 can be used.*

1.3 SPI emulator functional description

1.3.1 General description

Data transmission and reception is provided contemporary on two separate data unidirectional lines (MOSI, MISO) synchronized by common clock signal line provided by master. No addressing or acknowledgment control is implemented. Then, the SPI emulator is connected to external devices through 3 pins:

- MISO: Master In / Slave Out data. This pin can be configured through any I/O pin and can be used to transmit data in slave mode and receive data in master mode.
- MOSI: Master Out / Slave In data. This pin can be configured through any I/O pin and can be used to transmit data in master mode and receive data in slave mode.
- SCK: Serial Clock output for SPI master and input for SPI slave. This pin is configured as alternate function for timer channel 1 or channel 2.

1.3.2 Clock phase and clock polarity

Concerning SCK clock signal, the user has to respect fixed clock phase and polarity between communicating nodes. Four possible timing relationships may be chosen by software, using CPOL (clock polarity) and CPHA (clock phase) parameters.

The CPOL parameter controls the steady state value of the clock when no data is being transferred. This parameter affects both master and slave modes. If CPOL is low, the SCK pin has a low-level idle state. If CPOL is high, the SCK pin has a high-level idle state.

If the CPHA parameter is configured as 1EDGE, the first edge on the SCK pin (rising edge if CPOL is low, falling edge if CPOL is high) is the MSBit capture strobe. Data are latched on the occurrence of the first clock transition.

If the CPHA parameter is configured as 2EDGE, the second edge on the SCK pin (rising edge if CPOL is high, falling edge if CPOL is low) is the MSBit capture strobe. Data are latched on the occurrence of the second clock transition.

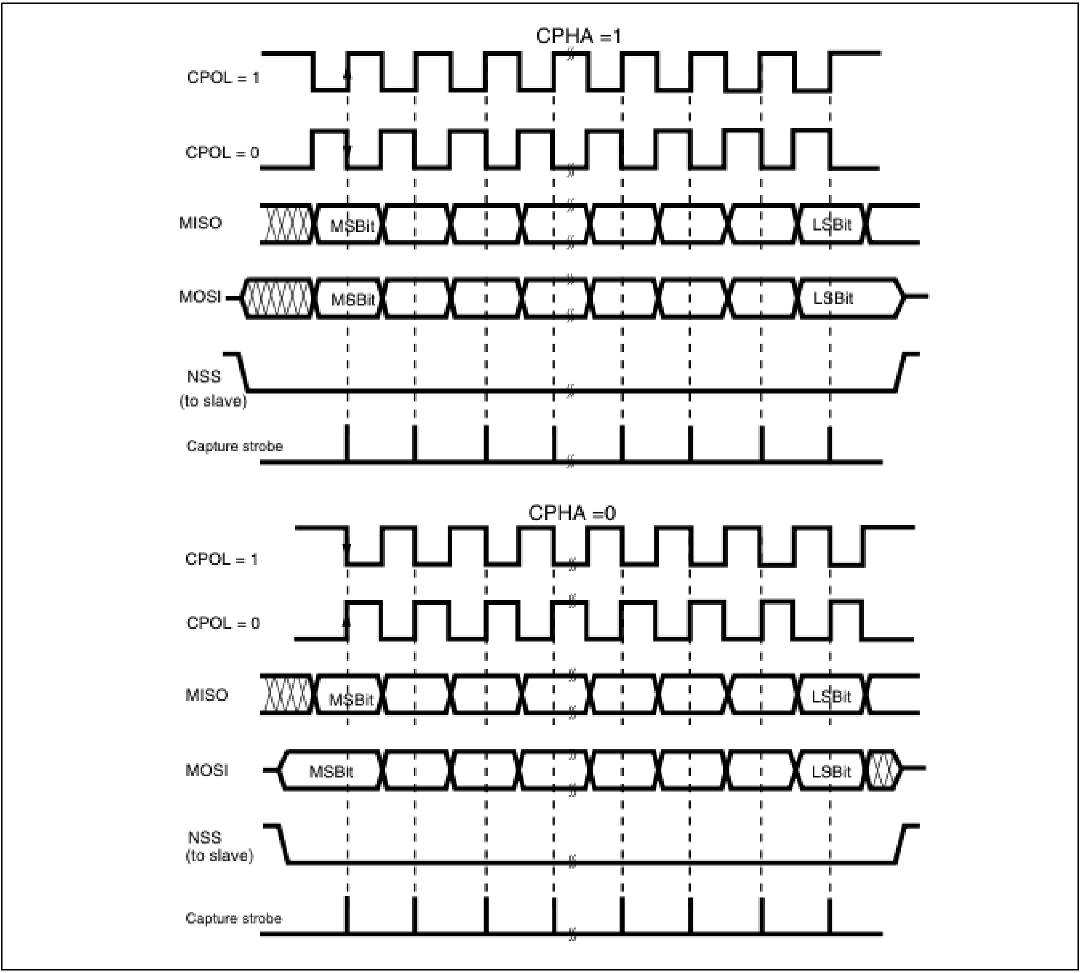
The combination of the CPOL and CPHA parameter selects the data capture clock edge (see *Table 2*).

Table 2. CPOL and CPHA configurations

CPOL	CPHA	Action
0	0	Data output on the rising edge of SCK. Input data is latched on the falling edge.
0	1	Data output one half-cycle before the first rising edge of SCK and on subsequent falling edges. Input data is latched on the rising edge of SCK.
1	0	Data iutput on the falling edge of SCK. Input data is latched on the rising edge.
1	1	Data output one half-cycle before the first falling edge of SCK and on subsequent rising edges. Input data is latched on the falling edge of SCK.

Figure 2 shows an SPI transfer with the four combinations of the CPHA and CPOL. The diagram may be interpreted as a master or slave timing diagram where the SCK pin, the MISO pin, the MOSI pins are directly connected between the master and the slave device.

Figure 2. Data clock timing diagram



1.3.3 Data frame format

The number of bits to be transacted is fixed between nodes but arbitrarily configurable. Each data frame is 8 or 16 bits long depending on user configuration and can be formatted either MSB-first or LSB-first. The selected data frame format is applicable for transmission and/or reception.

1.3.4 Configuring the SPI in master mode

Transmission process control is based on regular interrupts from the timer overflow events to DMA. Data for transmission must be firstly formatted by CPU and stored into dedicated variable (transmit buffer).

In this configuration, master starts the flow, handles clock and data signal.

The transmission sequence includes the following steps:

Single-frame transmission

1. The timer is configured in PWM mode to generate the clock signal with a frequency determined by the value of the TIMx_ARR register, and a duty cycle determined by the value of the TIMx_CCRx register.
2. The CPU formats the frame to be sent to the memory according to the First Bit configuration.
3. The timer sends a request to the DMA at each period of the clock signal to transfer one bit from memory to MOSI pin.

Once the frame transmission is complete, the timer stops generating the clock signal and the TxC flag (SPI transmission complete) is set.

Multiple-frame transmission

Multiple-frame transmission is based on a FIFO buffer with a threshold level of ½, this means that the buffer is effectively divided into two equal halves so data in one half can be transferred by DMA while data is being formatted by CPU in the second half. This allows the CPU to process one memory area while the second memory area is being used by the DMA transfer. At each end of transaction a half transfer complete or a transfer complete interrupt is generated by the DMA to ensure that the CPU swaps from one memory target to another. This operation is repeated until all frames are transmitted. When this is done, the timer stops generating the clock signal and the TxC flag is set.

Single-frame reception

The reception sequence includes the following steps:

1. The CPU checks if the SPI is ready and the RX buffer is empty.
2. The timer is configured in PWM mode to generate the clock signal with a frequency determined by the value of the TIMx_ARR register, and a duty cycle determined by the value of the TIMx_CCRx register.
3. The timer sends a request to the DMA at each period of the clock signal to transfer one bit from MISO pin to the memory.
4. The CPU formats the received frame.

Once the frame reception is complete, the timer stops generating the clock signal and the RxC flag (SPI reception complete) is set.

Multiple-frame reception

DMA transmits data to the memory. When the first half of the FIFO buffer is filled, a half transfer complete interrupt is generated by the DMA so the CPU formats data and stores them in SRAM. At the same time the DMA continues to fill the second half. When this is done, a transfer complete interrupt is generated and the CPU formats this data. The DMA configured in circular mode returns to the initial pointer and keeps going. This operation is repeated until all frames are received. Once the frame reception is complete, the timer stops generating the clock signal and the RxC flag is set.

1.3.5 Configuring the SPI in slave mode

In the slave configuration, the serial clock is received on the SCK pin from the master device.

The timer is configured in input capture mode, so the transmission process starts when the clock signal is detected on the timer input channel.

Single-frame transmission

The transmission sequence includes the following steps:

1. The CPU formats the frame to be sent to the memory according to the First Bit configuration.
2. The timer sends a request to the DMA at each period of the clock signal to transfer one bit from memory to MISO pin. The requests are programmed to occur at the rising or falling edge of the input signal, depending on CPHA configuration.
3. Once the frame transmission is complete, the TxC flag (SPI transmission complete) is set.

Multiple-frame transmission

Multiple-frame transmission is based on a FIFO buffer with a threshold level of ½, this means that the buffer is effectively divided into two equal halves so data in one half can be transferred by the DMA while data is being formatted by the CPU in the second half. This allows the CPU to process one memory area while the second memory area is being used by the DMA transfer. At each end of transaction a half transfer complete or a transfer complete interrupt is generated by the DMA to ensure that the CPU swaps from one memory target to another. This operation is repeated until all frames are transmitted. When this is done, the timer stops generating the clock signal and the TxC flag is set.

Single-frame reception

The transmission sequence includes the following steps:

1. The CPU checks if the SPI is ready and the RX buffer is empty.
2. On the rising (or falling) edge of the external trigger, the timer generates a DMA request. As the GPIO data register address is set to DMA peripheral address, the DMA controller reads the data from the GPIO port on each DMA request, and stores it into an SRAM buffer.
3. The CPU formats the received frame.
4. Once the frame reception is complete, the RxC flag (SPI reception complete) is set.

Multiple-frame reception

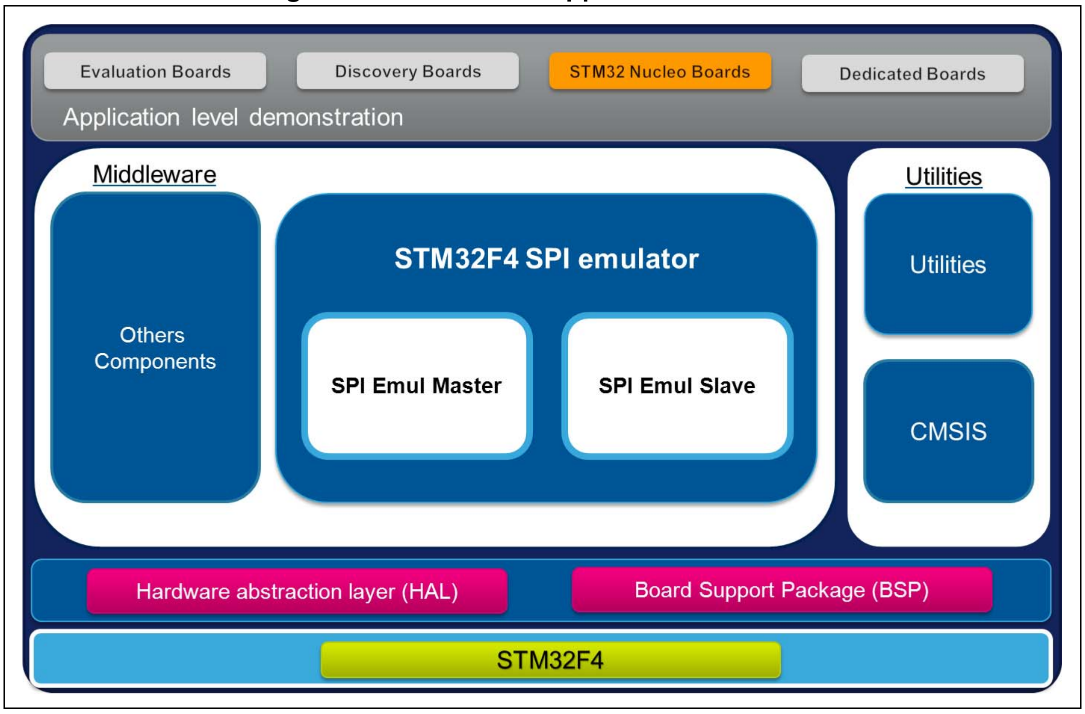
DMA transmits data to the memory. When the first half of the FIFO buffer is filled, a half transfer complete interrupt is generated by the DMA so the CPU can format data and store them in the SRAM. The DMA continues to fill the second half, when it’s done, a transfer complete interrupt is generated and the CPU formats these data. The DMA configured in circular mode returns to the initial pointer and keeps going. This operation is repeated until all frames are received. When this is done, the timer stops generating the clock signal and the RxC flag is set.

2 Software description

2.1 Implementation structure

STM32 SPI emulator package is based on STM32 Cube architecture. *Figure 3* shows how the package is structured internally and how it can be implemented in a complete project.

Figure 3. SPI emulator application level view



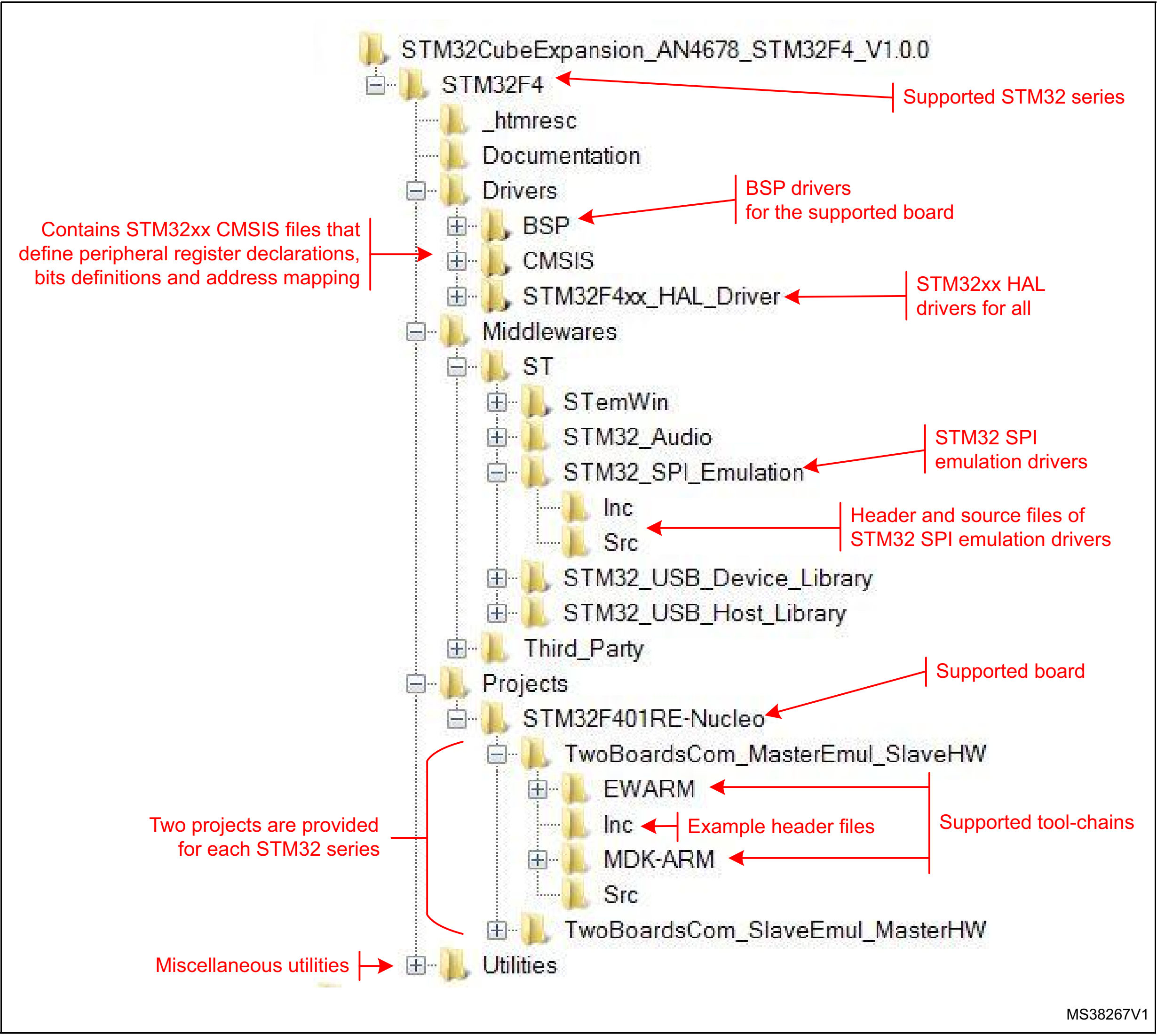
STM32 SPI emulation package is located in middlewares level and support STM32F4 series. It is based on modular architecture that means other STM32 series can be supported without any impact on the current implementation.

In Application layer, STM32 SPI emulation package provide a set of examples for the most common development tools.

2.2 Package organization

The application note is supplied in a zip file. The extraction of the zip file generates one folder, STM32CubeExpansion_AN4678_STM32F4_V1.0.0, which contains the subfolders shown in *Figure 4*.

Figure 4. STM32CubeExpansion V1.0.0 package organization


STM32CubeExpansion_AN4678_STM32F4_V1.0.0
STM32F4
_htmresc
Documentation
Drivers
BSP
CMSIS
STM32F4xx_HAL_Driver
Middlewares
ST
STemWin
STM32_Audio
STM32_SPI_Emulation
Inc
Src
STM32_USB_Device_Library
STM32_USB_Host_Library
Third_Party
Projects
STM32F401RE-Nucleo
TwoBoardsCom_MasterEmul_SlaveHW
EWARM
Inc
MDK-ARM
Src
TwoBoardsCom_SlaveEmul_MasterHW
Utilities
Supported STM32 series
BSP drivers for the supported board
Contains STM32xx CMSIS files that define peripheral register declarations, bits definitions and address mapping
STM32xx HAL drivers for all
STM32 SPI emulation drivers
Header and source files of STM32 SPI emulation drivers
Supported board
Two projects are provided for each STM32 series
Example header files
Supported tool-chains
Miscellaneous utilities
MS38267V1

2.3 Transmission

2.3.1 Format data procedure

This solution integrates a FIFO structures buffering for transmitter.

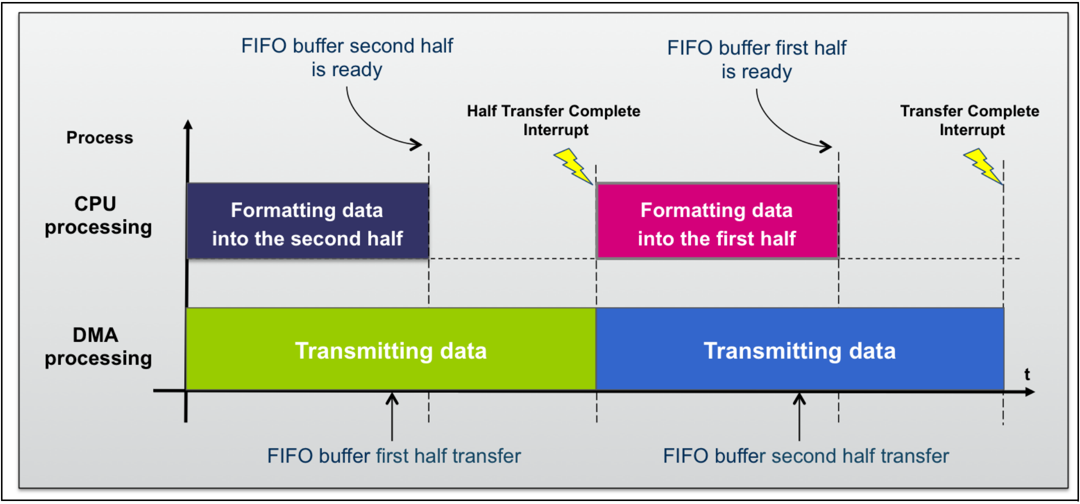
The FIFO is used to temporarily store data formatted by CPU before transmitting them to the destination. Also DMA is configured in circular mode to handle continuous data flows (the DMA_SxNDTR register is then reloaded automatically with the previously programmed value). The FIFO structure implemented helps to:

- reduce SRAM access and so give more time for the other peripherals to access the bus matrix without additional concurrency;
- allow software to do burst transactions which optimize the transfer speed and bandwidth.

Data package can be prepared in advance by the CPU and stored into an SRAM buffer. This FIFO buffer can host 20 frames, and the threshold level is ½, so while DMA channel is transferring data from the first half, CPU prepares the next block of data to be send in the second half and vice versa. This is managed by the half transfer complete and transfer complete interrupts generates by DMA after each transfer of 10 frames.

The principle of formatting and sending a given number of bytes using CPU and DMA in Tx mode is shown in *Figure 5*.

Figure 5. STM32CubeExpansion V1.0.0 package organization

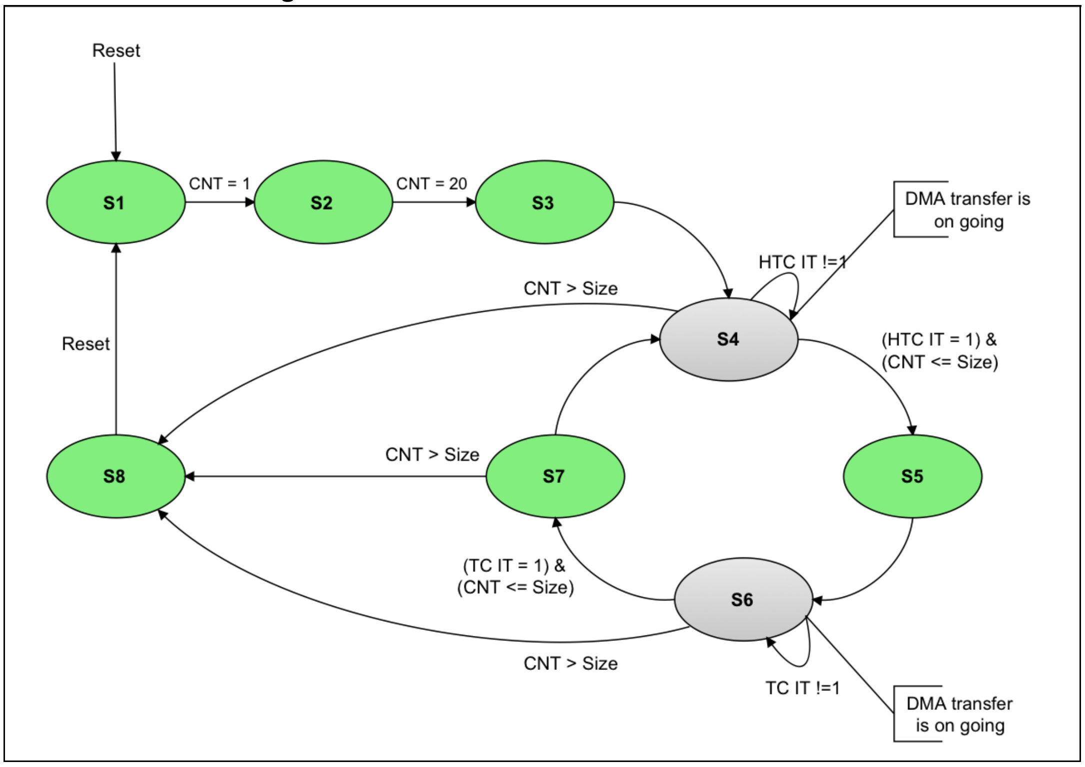

FIFO buffer second half is ready
FIFO buffer first half is ready
Half Transfer Complete Interrupt
Transfer Complete Interrupt
Process
CPU processing
Formatting data into the second half
Formatting data into the first half
DMA processing
Transmitting data
Transmitting data
t
FIFO buffer first half transfer
FIFO buffer second half transfer

2.3.2 Transmission routine

Figure 6 gives an overview of the SPI emulator transmission routine, in it we have:

- S1: Initialize parameters and configure the SPI emulator
- S2: Format data in the whole FIFO buffer
- S3: Start transmission process by enabling DMA in circular mode
- S4: CPU in mode sleep while DMA is transferring data from the first half of the FIFO buffer Data_Buffer_Tx
- S5: CPU formats data in the first half of the FIFO buffer Data_Buffer_Tx after DMA half transfer complete interrupt
- S6: CPU in mode sleep while DMA is transferring data from the second half of the FIFO buffer Data_Buffer_Tx
- S7: CPU formats data in the second half of FIFO buffer Data_Buffer_Tx after DMA transfer complete interrupt
- S8: End of transmission
- CNT: TxXferCount is a counter that is incremented after the completion of each DMA transfer.
- HTC IT: DMA Half Transfer Complete interrupt.
- TC IT: DMA Transfer Complete interrupt.

Figure 6. Transmission routine state machine



2.3.3 Peripheral settings

- GPIO:
  - BSRR is used as destination register for DMA transfers.
- DMA2:
  - The transfer is performed by words.
  - Channel6 and Stream1 are used for transmission.
  - DMA half transfer complete and transfer complete interrupts are used at the end of data package transfers.
- Timer 1:
  - Channel1 is configured as capture compare for DMA transmit requests.
  - Channel2 is configured in PWM mode for clock generation in master mode.
- SRAM
  - An SRAM buffer is used to format data in Tx mode:
  - In case of 8 bits data length, Data_Buffer_Tx[8bits x 20frames]
  - In case of 16 bits data length, Data_Buffer_Tx[16bits x 20frames]

2.4 Reception

2.4.1 Format data procedure

This solution integrates a FIFO structures buffering for receiver.

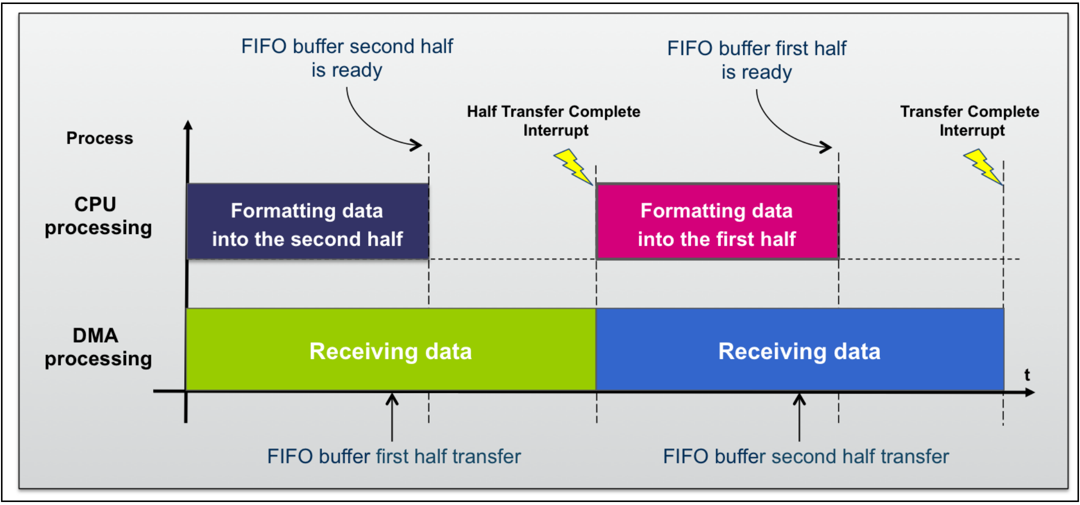
The FIFO is used to temporarily store data transmitted by DMA before formatting them by CPU. DMA is configured in circular mode to handle continuous data flows (the DMA_SxNDTR register is then reloaded automatically with the previously programmed value). The FIFO structure implemented helps to:

- reduce SRAM access and so give more time for the other peripherals to access the bus matrix without additional concurrency,
- allow software to do burst transactions which optimize the transfer speed and bandwidth.

Data package transmitted by DMA is stored into a FIFO buffer and can be formatted by CPU. This FIFO buffer can host 20 frames, and the threshold level is ½, so while DMA channel is transferring data to the first half, CPU formats the received block of data in the second half and vice versa. This is managed by the half transfer complete and transfer complete interrupts generates by DMA after each transfer of 10 frames.

The principle of formatting and receiving a given number of bytes using CPU and DMA in Rx mode is shown in *Figure 7*.

Figure 7. Formatting and receiving data using CPU and DMA

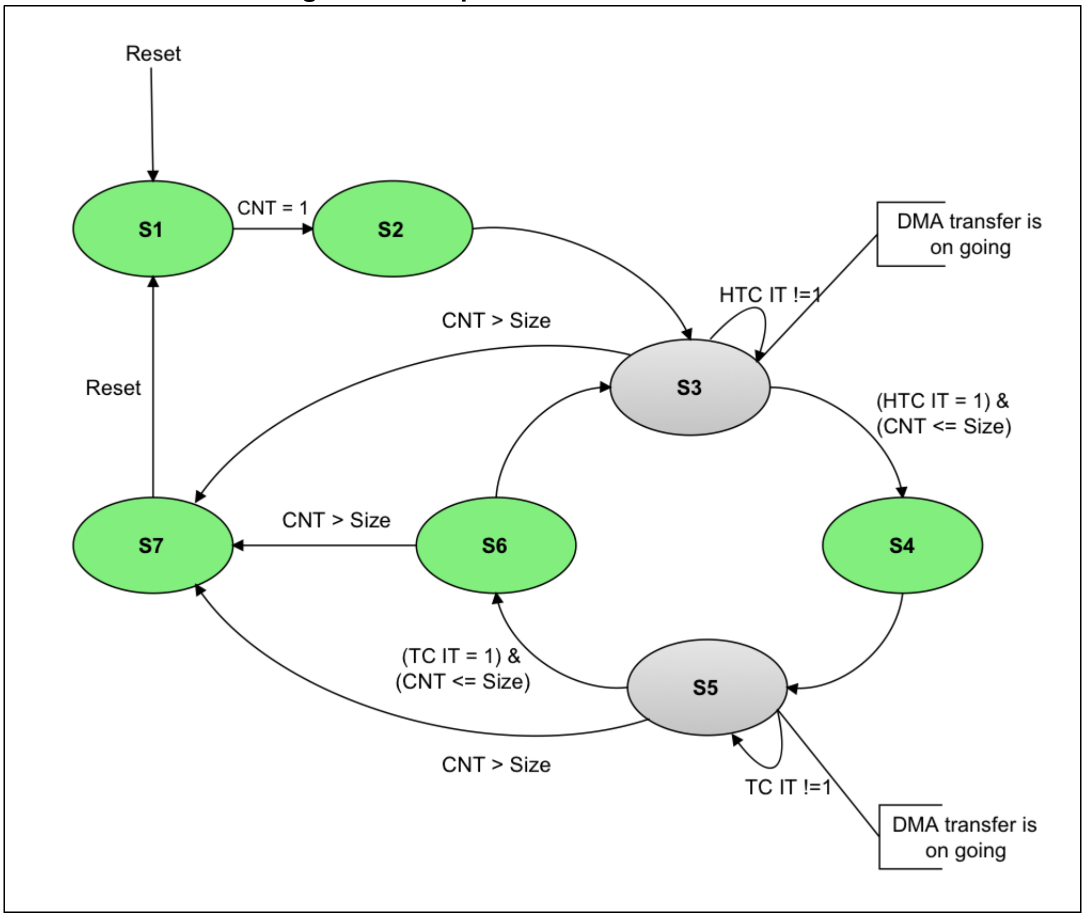

FIFO buffer second half is ready
FIFO buffer first half is ready
Half Transfer Complete Interrupt
Transfer Complete Interrupt
Process
CPU processing
Formatting data into the second half
Formatting data into the first half
DMA processing
Receiving data
Receiving data
t
FIFO buffer first half transfer
FIFO buffer second half transfer

2.4.2 Transmission routine

Figure 8 gives an overview of the SPI emulator reception routine, in it we have:

- S1: Initialize parameters and configure the SPI emulator
- S2: Start reception process by enabling DMA in circular mode
- S3: CPU in mode sleep while DMA is transferring data to the first half of FIFO buffer Data_Buffer_Rx
- S4: CPU formats data in the first half of FIFO buffer Data_Buffer_Tx after DMA half transfer complete interrupt
- S5: CPU in mode sleep while DMA is transferring data to the second half of FIFO buffer Data_Buffer_Rx
- S6: CPU formats data in the second half of Data_Buffer_Tx after DMA transfer complete interrupt
- S7: End of transmission
- CNT: TxXferCount is a counter that is incremented after the completion of each DMA transfer.
- HTC IT: DMA Half Transfer Complete interrupt.
- TC IT: DMA Transfer Complete interrupt.

Figure 8. Reception routine state machine



2.4.3 Peripheral settings

- GPIO:
  - IDR is used as source register for DMA transfers.
- DMA2:
  - The transfer is performed by words.
  - Channel6 and Stream2 are used for reception.
  - DMA half transfer complete and transfer complete interrupts are used at the end of data package transfers.
- Timer 1:
  - Timer1 Channel2 is configured as capture compare for DMA transmit requests.
  - Timer1 Channel2 is configured in PWM mode for clock generation in master mode.
- SRAM
  - An SRAM buffer is used to format data in Rx mode:
  - In case of 8 bits data length , Data_Buffer_Rx[8bits x 20frames]
  - In case of 16 bits data length, Data_Buffer_Rx[16bits x 20frames]

2.5 SPI emulator API

This section provides a set of functions ensuring SPI emulation.

Table 3. SPI Emulation functions

Function name	Description
HAL_SPI_Emul_Init	Initialize the SPI Emulation according to the specified parameters in the *SPI_Emul_InitTypeDef* and create the associated handle.
HAL_SPI_Emul_Transmit_DMA	Transmit an amount of data in no-blocking mode with DMA
HAL_SPI_Emul_Receive_DMA	Receive an amount of data in no-blocking mode with DMA
HAL_SPI_Emul_TransmitReceive_DMA	Transmit and Receive an amount of data in no-blocking mode with DMA

2.5.1 HAL_SPI_Emul_Init function

Table 4. HAL_SPI_Emul_Init

Function name	***HAL_SPI_Emul_Init***
Prototype	*HAL_StatusTypeDef HAL_SPI_Emul_Init(SPI_Emul_HandleTypeDef *hspi)*
Behavior	Initializes the SPI Emulation according to the specified parameters in the *SPI_Emul_InitTypeDef* and creates the associated handle.

Table 4. HAL_SPI_Emul_Init (continued)

Function name	***HAL_SPI_Emul_Init***
Parameter	**hspi*: pointer to *SPI_Emul_HandleTypeDef* structure that contains the configuration information for SPI emulator
Return value	HAL status

2.5.2 HAL_SPI_Emul_InitTypeDef structure

Table 5. HAL_SPI_Emul_Init

Field name	Description
uint32_t Mode	Specifies the SPI operating mode
uint32_t Direction	Specifies the SPI Directional mode state
uint32_t DataSize	Specifies the SPI data size
uint32_t CLKPolarity	Specifies the serial clock steady state
uint32_t CLKPhase	Specifies the clock active edge for the bit capture
uint32_t SPI_Clk	Specifies the SPI Clock Frequency
uint32_t FirstBit	Specifies whether data transfers start from MSB or LSB bit
uint32_t RxPinNumber	Specifies the number of Receiver Pin
uint32_t TxPinNumber	Specifies the number of Transmitter Pin
uint32_t ClkPinNumber	Specifies the number of Clock Pin

2.5.3 HAL_SPI_Emul_Transmit_DMA function

Table 6. HAL_SPI_Emul_Transmit_DMA

Function name	***HAL_SPI_Emul_Transmit_DMA***
Prototype	*HAL_StatusTypeDef HAL_SPI_Emul_Transmit_DMA (SPI_Emul_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size)*
Behavior	Transmit in simplex mode an amount of data in no-blocking mode with DMA
Parameter	– **hspi*: pointer to *SPI_Emul_HandleTypeDef* structure that contains the configuration information for SPI emulator – **pData*: Pointer to data buffer – *Size*: Amount of data to be sent
Return value	HAL status

2.5.4 HAL_SPI_Emul_Receive_DMA function

Table 7. HAL_SPI_Emul_Receive_DMA

Function name	*HAL_SPI_Emul_Receive_DMA*
Prototype	*HAL_StatusTypeDef HAL_SPI_Emul_Receive_DMA (SPI_Emul_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size)*
Behavior	Receive in simplex mode an amount of data in no-blocking mode with DMA
Parameter	– **hspi*: pointer to *SPI_Emul_HandleTypeDef* structure that contains the configuration information for SPI emulator – **pData*: Pointer to data buffer – *Size*: Amount of data to be sent
Return value	HAL status

2.5.5 HAL_SPI_Emul_TransmitReceive_DMA function

Table 8. HAL_SPI_Emul_TransmitReceive_DMA

Function name	*HAL_SPI_Emul_TransmitReceive_DMA*
Prototype	*HAL_StatusTypeDef HAL_SPI_Emul_TransmitReceive_DMA (SPI_Emul_HandleTypeDef *hspi, uint8_t *pData, uint16_t Size)*
Behavior	Transmit and Receive in simplex mode an amount of data in no-blocking mode with DMA
Parameter	– **hspi*: pointer to *SPI_Emul_HandleTypeDef* structure that contains the configuration information for SPI emulator – **pData*: Pointer to data buffer – *Size*: Amount of data to be sent
Return value	HAL status

2.5.6 Callback functions

Callback functions are also available, as shown in *Table 9*. They allow the user to implement his own code in the user file.

Table 9. HAL_SPI_Emul_TransmitReceive_DMA

Function name	Parameters	Description
*__weak void HAL_SPI_Emul_RxCpltCallback (SPI_Emul_HandleTypeDef *hspi)*	*hspi*: SPI emulator handle	This function is called at the end of the reception process
*__weak void HAL_SPI_Emul_RxHalfCpltCallback (SPI_Emul_HandleTypeDef *hspi)*		This function is called at the half of the reception process
*__weak void HAL_SPI_Emul_TxCpltCallback (SPI_Emul_HandleTypeDef *hspi)*		This function is called at the end of the transmission process
*__weak void HAL_SPI_Emul_TxHalfCpltCallback (SPI_Emul_HandleTypeDef *hspi)*		This function is called at half of the transmission process
*__weak void HAL_SPI_Emul_ErrorCallback (SPI_Emul_HandleTypeDef *hspi)*		This function is called when a communication error is detected.

3 Example

The example illustrates a data exchange between the SPI emulator and the hardware SPI.

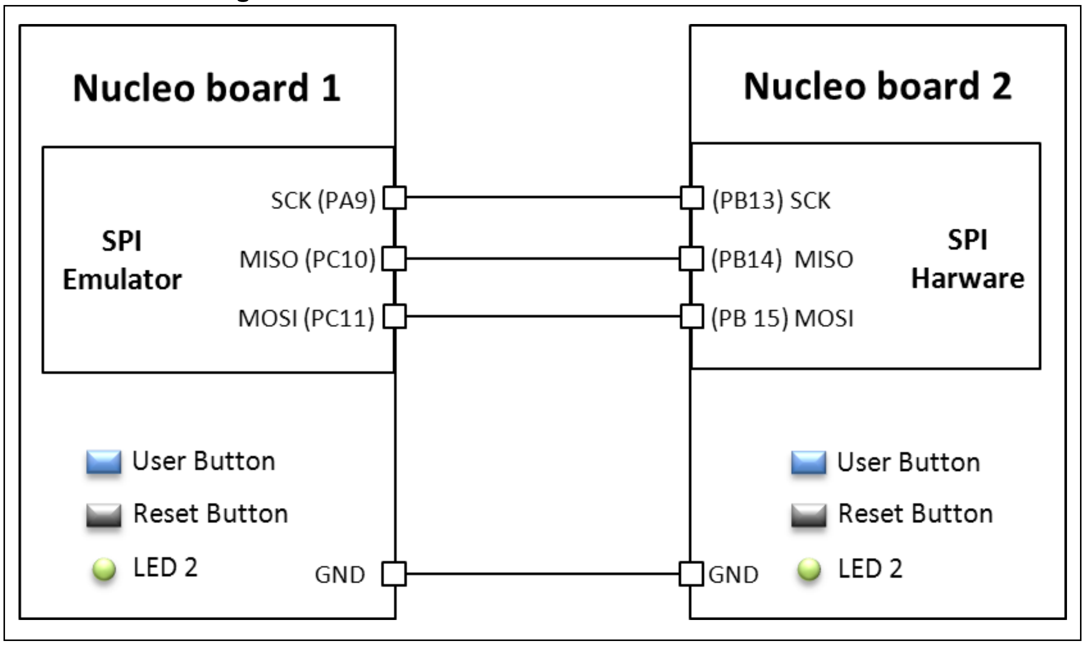
3.1 Hardware requirements

The hardware required to run this example is the following:

- Two Nucleo boards (NUCLEO-F401RE)
- Two Mini-USB cables to power the boards and to connect the Nucleo embedded ST-LINK for debugging and programming.

The connection between the two Nucleo boards through SPI lines is described in *Figure 9*.

Figure 9. SPI emulator and SPI hardware connection



3.2 Software settings

In this firmware package two project examples are provided, each one includes two workspaces:

1. SPI_EMUL_MasterSide and SPI_HW_SlaveSide in the first example called TwoBoardsCom_MasterEmul_SlaveHW, as shown in *Figure 10*
2. SPI_EMUL_SlaveSide and SPI_HW_MasterSide in the second example called TwoBoardsCom_SlaveEmul_MasterHW, as shown in *Figure 11*.

Figure 10. Example MDK-ARM™ workspaces (SPI Master emul / Slave HW)

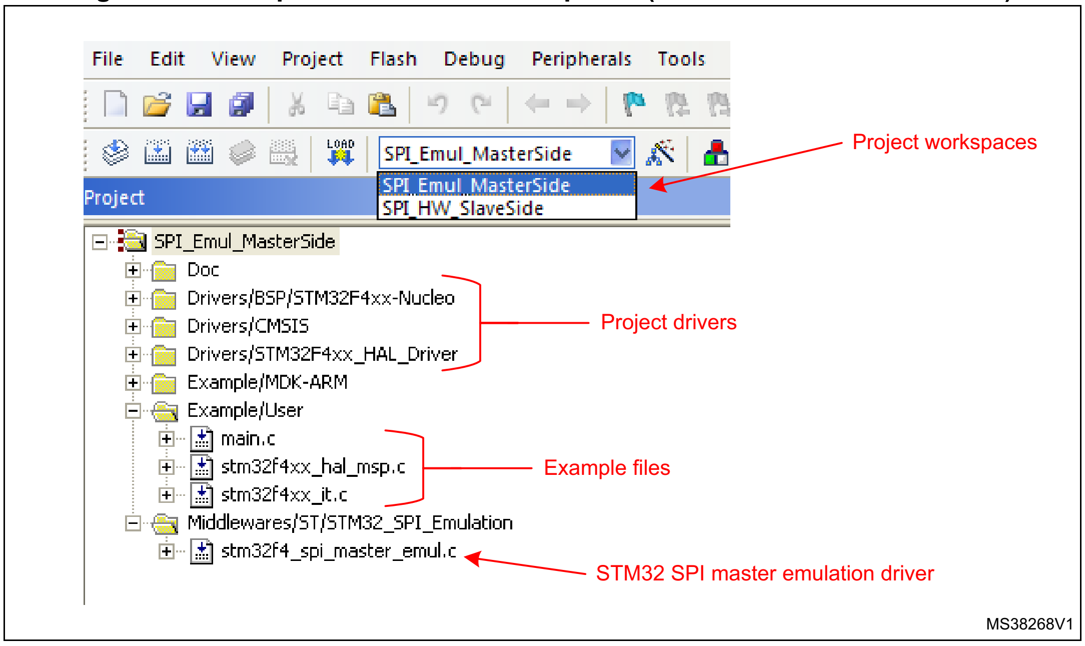
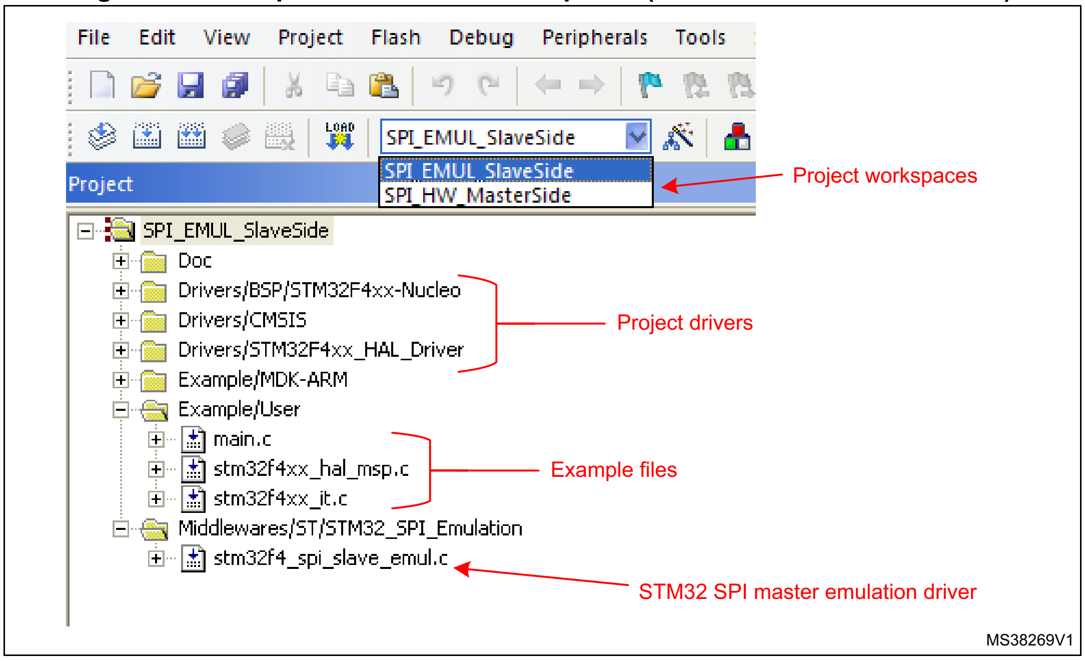


Figure 11. Example MDK-ARM™ workspaces (SPI Slave emul / Master HW)



To make the program work, follow the steps described below:

1. Open your preferred toolchain (EWARM or MDK-ARM™).
2. Rebuild all files and load your image into target memory.
3. Run the example.

3.3 Running the example

To run the example, follow the sequence below:

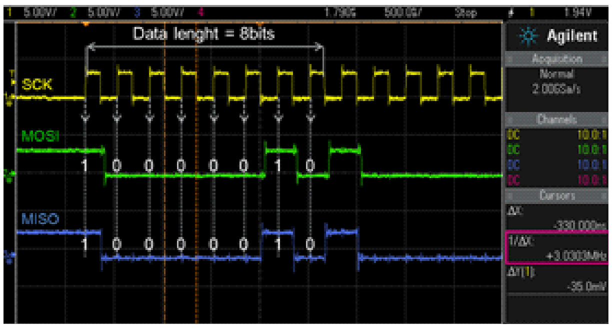
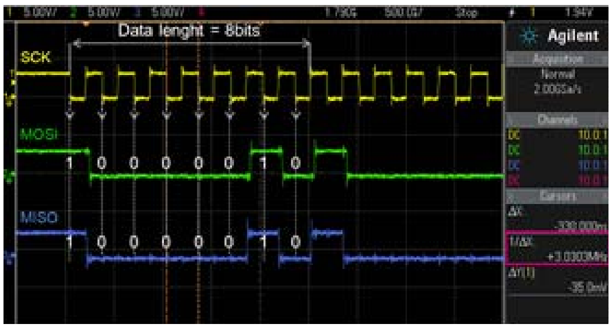
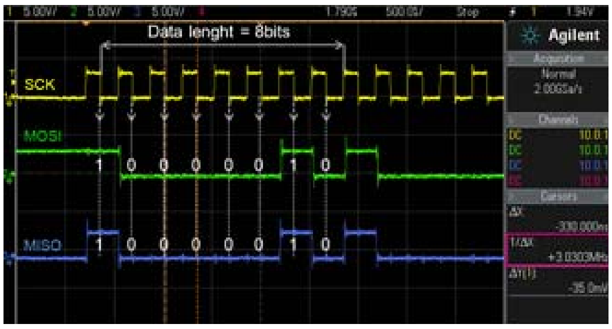
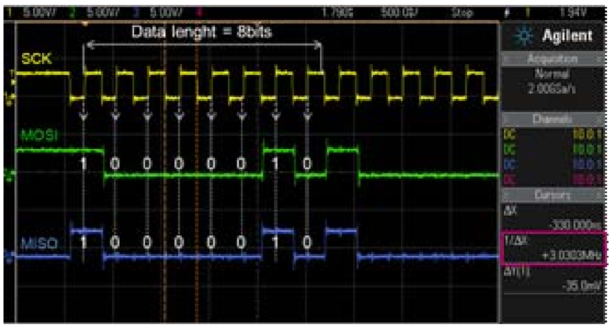
1. Power on the two boards.
2. Load the code in each board MCU.
3. Press the user button key on board 1. The example then starts running and the SPI emulator starts transmitting data.
4. The SPI hardware receives the data and sends them back to SPI emulator.
5. The data transmitted by the SPI emulator is compared to received ones: if data do not match, the green LED (LED2) toggles continuously.

Note: For more details, refer to the readme.txt inside the firmware package.

3.4 Frame waveforms

Table 10 shows examples six different configurations of SPI emulator full duplex ‘A’ (0100 0001) character transfer.

Table 10. Examples of configurations of SPI Emulator full duplex

Case 1 – Clock speed = 3 MHz – Data length = 8 bits – Data order = LSB first – CPOL = 0, CPHA = 0 	**Case 2** – Clock speed = 3 MHz – Data length = 8 bits – Data order = LSB first – CPOL = 1, CPHA = 0 
Case 3 – Clock speed = 3 MHz – Data length = 8 bits – Data order = LSB first – CPOL = 0, CPHA = 1 	**Case 4** – Clock speed = 3 MHz – Data length = 8 bits – Data order = LSB first – CPOL = 1, CPHA = 1 

4 SPI emulator CPU load and footprint

The SPI emulator uses the CPU for frames processing which includes:

- formatting data
- saving data in internal SRAM
- handling DMA interrupts

4.1 CPU load

As discussed above, during data transfer and reception, CPU is used for frames processing, this action requires the time defined in *Table 9*.

Table 11. Clock cycles needed for frame processing

Data size		8 bits	16 bits
Mode	Transmission	116 × CPU clock cycles per frame	225 × CPU clock cycles per frame
	Reception	108 × CPU clock cycles per frame	210 × CPU clock cycles per frame

The calculation of CPU load is based on the following formula:

$$\mathrm{CPUload} = \frac{\frac{\mathrm{CPUclockcycles}}{\mathrm{SystemClk}} \times \mathrm{SPIfreq}}{\mathrm{Datasize}} \times 100$$

Example:

The software settings used to obtain the results given in *Figure 12* are the following:

- System clock: 84 MHz
- Data size: 8 bits
- SPI clock: 1 MHz

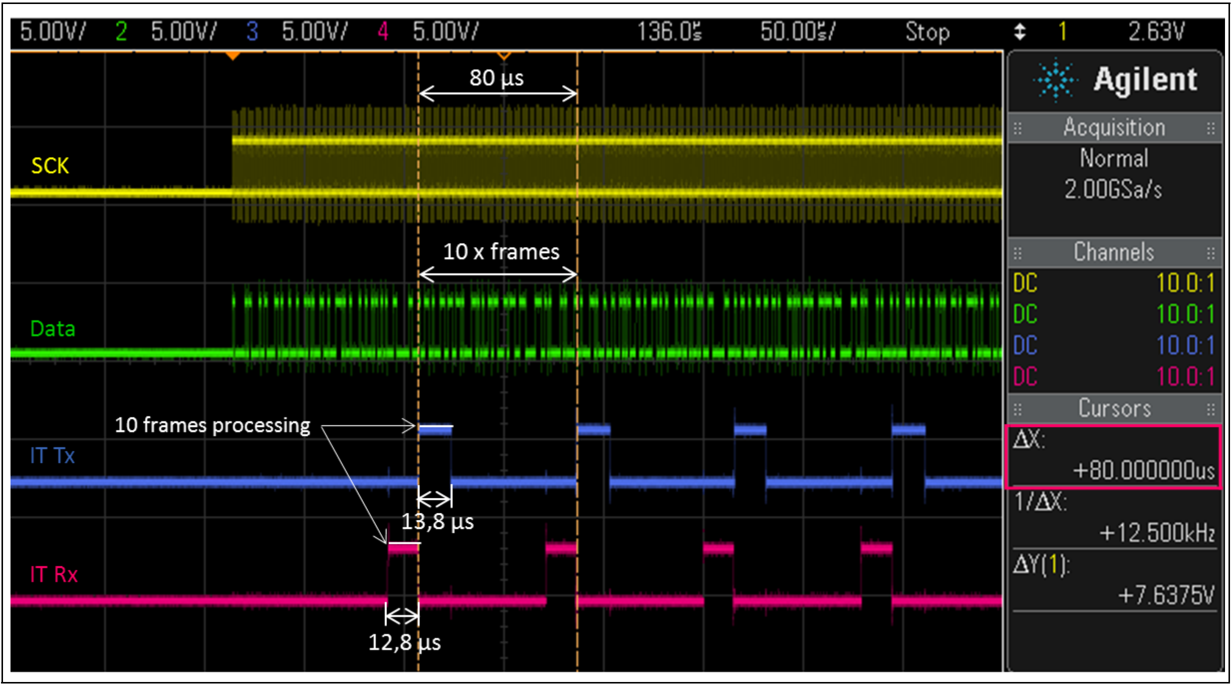
The program was compiled with MDK-ARM™ V5.14, optimization level3 (-O3) for speed, so, applying the above formula

CPU load Tx = $((116 / (84 \times 10^6)) \times 10^6) / 8 \times 100$ =17.25 %

CPU load Rx = $((108 / (84 \times 10^6)) \times 10^6) /8 \times 100$ = 16.0%

CPU load full duplex= 17.25 % + 16.0 % = 33.25 %.

Figure 12. SPI emulator CPU load



4.2 SPI emulator memory footprint

User can easily adapt the example to his own application as only a small amount of code size is required by the SPI program.

Table 12 gives an estimate of the code size required by the SPI emulator compiled with MDK-ARM™ V5.14, optimization level3 (-O3) for speed.

Table 12. SPI emulator memory footprint

Mode	Direction	Flash memory footprint (bytes)	RAM footprint (bytes)	
			8 bits	16 bits
Master	Full duplex	4394	1556	2836
	Transmission	3750	916	1576
	Reception	3544	916	1576
Slave	Full duplex	4230	1532	2812
	Transmission	3554	892	1548
	Reception	3388	892	1548

5 Conclusion

This application note demonstrates the implementation of an effective emulation of Serial Peripheral interface (SPI), which can increase virtually the number of serial communication peripherals in STM32 microcontrollers and improve their capability.

User can benefit from this additional feature, provided he respects the limitations and considers a reasonable performance bandwidth.

6 Revision history

Table 13. Document revision history

Date	Revision	Changes
04-Aug-2015	1	Initial release.